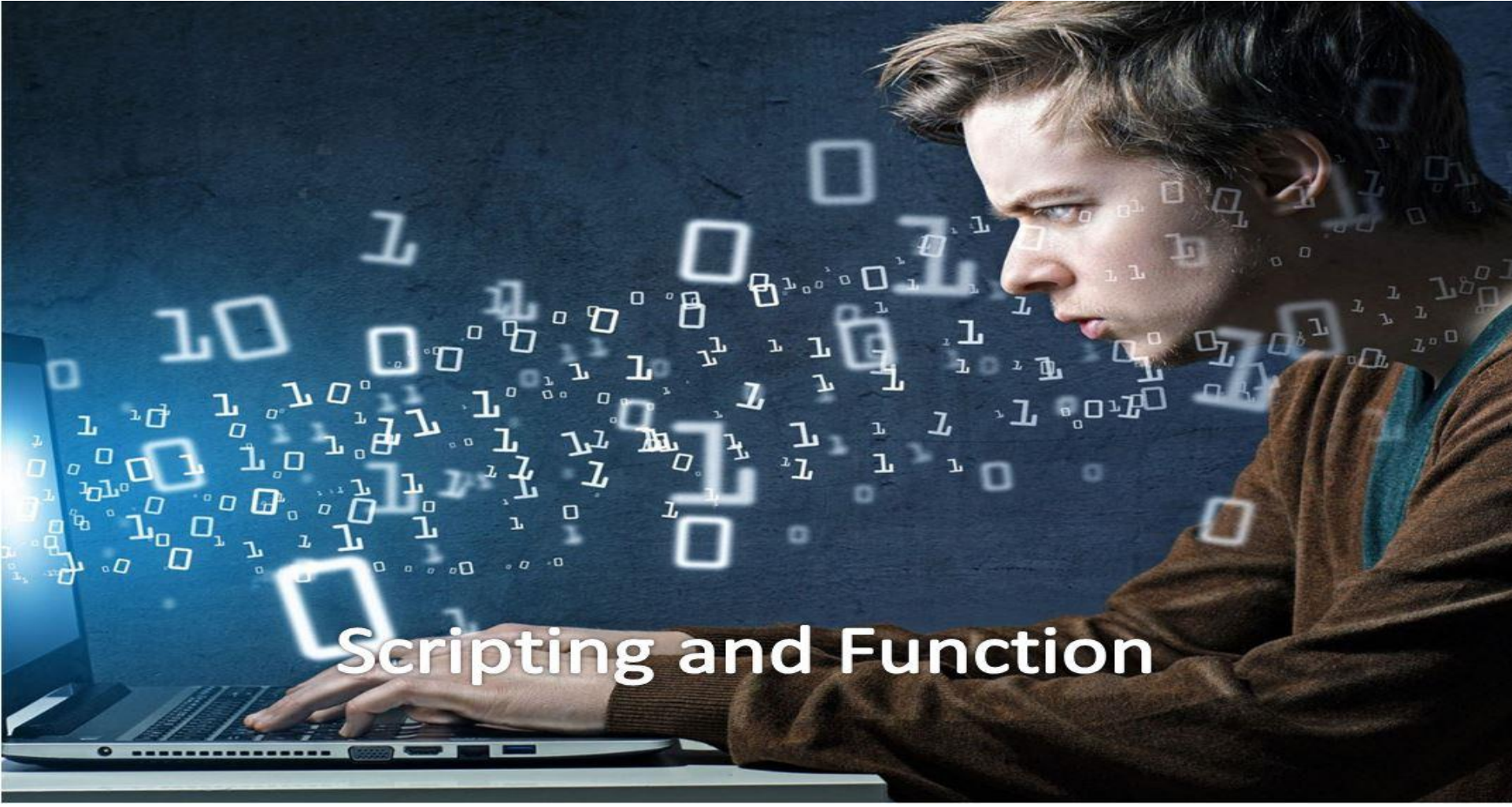


Kok Woon Chee
woonchee.kok@gmail.com

UNIMAS
UNIVERSITI MALAYSIA SARAWAK

FoCuS IT
focus on individual transformation

Outline

- Introduction of function
- User defined function

Function

- Incorporate sets of instructions that user want to use repeatedly or that,
- Because of their complexity, are better self-contained in a sub-program and called when needed.

Function (1)

- Is a code written to carry out a specified task; it may accept arguments or parameter and it may return one or more values.

- Argument: Inputs
- Return Value: outputs

Example of a function

```
pow <- function(x, y)
   {
   # function to print x raised to the power y

   result <- x^y
   print(paste(x,"raised to the
power",y,"is",result))
}
```

Here we have created a function called pow(). It takes two arguments, finds the first argument raised to the power of second argument and prints the result in appropriate format.
We have used a built-in function paste() which is used to concatenate strings.

Function call

```
> pow(8,2)

> pow(2,4)
```

Call the above function as above:
Here the arguments used in the function declaration (x and y) are called formal arguments and those while calling the function are called actual arguments

User Defined Function (UDF)

- Need to accomplish a particular task and
- No existing library and function is available.

```
function.name<-function(arguments)
{
        computations on the arguments
}
```

Named Arguments

In the above function calls, the argument matching of formal argument to the actual arguments takes place in positional order. This means that, in the call pow(8,2), the formal arguments x and y are assigned 8 and 2 respectively

Named Arguments (1)

- We can also call the function using named arguments. When calling a function in this way, the order of the actual arguments doesn't matter.

```
> pow(8,2)
> pow(x=8,y=2)
> pow(y=2,x=8)
```

Named Arguments (2)

- We can also use named and unnamed arguments in a single call. In such case, all named arguments are matched first and then the remaining unnamed arguments are matched in a positional order

```
> pow(x=8,2)
[1] “8 raised to the power of 2 is 64”
> pow(2,x=8)
```

Named Arguments (3)

- We can assign default values to arguments in a function in R. This is done by providing an appropriate value to the formal argument in the function declaration.

```
pow <- function(x, y=2)
  {
  # function to print x raised to the power y

  result <- x^y
  print(paste(x,"raised to the
power",y,"is",result))
}
```

Named Arguments (4)

- The use of default value to an argument makes it optional when calling the function.

```
# x=3
>pow(3)
[1] “3 raised to the power of 2 is 9”

>pow(3,1)
[1] “3 raised to the power of 1 is 3”
```

Here, y is optional and will take the value of 2 when not provided.

- Step 1: Launch RStudio, get initial directory
- Step 2: set the current directory
- Step 3: Open new R script to create function
- Step 4: Save and source your function
- Step 5: Use your first user defined function!

Return value from function

- Many a times, we will require our functions to do some processing and return back the result.
- This is accomplished with the **return()** function in R

```
return(expression)
```

- The value returned from a function can be any valid object.

Example of return()

```
check<-function(x)
{
   if (x>0)
        {
     result<-”Positive”
      }
   else if(x<0)
      {
      result<-”Negative”
      }
   else
      {
      result<-”Zero”
      }
   return(result)

}
```

- An example which will return whether a given number is positive, negative or zero.

```
>check(1)
[1]”Positive”
>check(-10)
[1] “Negative”
Check(0)
[1] “Zero”
```

Function without return()

- If there are no explicit returns from a function, the value of the last evaluated expression is returned automatically in R. For example, the following is equivalent to the above function.

Function without return()

```
check<-function(x)
{
   if (x>0)
         {
     result<-”Positive”
      }
   else if(x<0)
      {
      result<-”Negative”
      }
   else
       {
       result<-”Zero”
       }
   result

}
```

- We generally use explicit return() functions to return a value immediately from a function. If it is not the last statement of the function, it will prematurely end the function bringing the control to the place from which it was called.

```
check<-function(x)
{
   if (x>0)
        {
          return(”Positive”)
       }
   else if(x<0)
       {
       return(”Negative”)
       }
   else
       {
       return(”Zero”)
       }
}
```

- Above ex., if x>0, the function immediately returns “Positive” without evaluating rest of the body.

Multiple Returns

- The return() function can return only a single object. If we want to return multiple values in R, we can use a list (or other objects) and

```
multi_return <- function()
{
  my_list <- list("color" = "red", "size" =
20, "shape" = "round")
  return(my_list)
}
```

- Here, we create a list my_listwith multiple elements and return this single list.

```
> a<-multi_return()
> a
$color
[1] "red"

$size
[1] 20

$shape
[1] "round"
```

Recursive Function

- A function that calls itself is called a recursive function. This special programming technique can be used to solve problems by breaking them into smaller and simpler sub-problems.

- Let us take the example of finding the factorial of a number. Factorial of a positive integer number is defined as the product of all the integers from 1 to that number. For example, the factorial of 5 (denoted as 5!) will be 1*2*3*4*5=120. This problem of finding factorial of 5 can be broken down into a sub-problem of multiplying the factorial of 4 with 5.

$$5!=5*4!$$

$$n!=n*(n-1)!$$

Example of a Recursive Function in R

```
recursive.factorial<-function(x)
{
    if (x==0)
     return (1)
    else
     return (x*recursive.factorial(x-1))
}
```

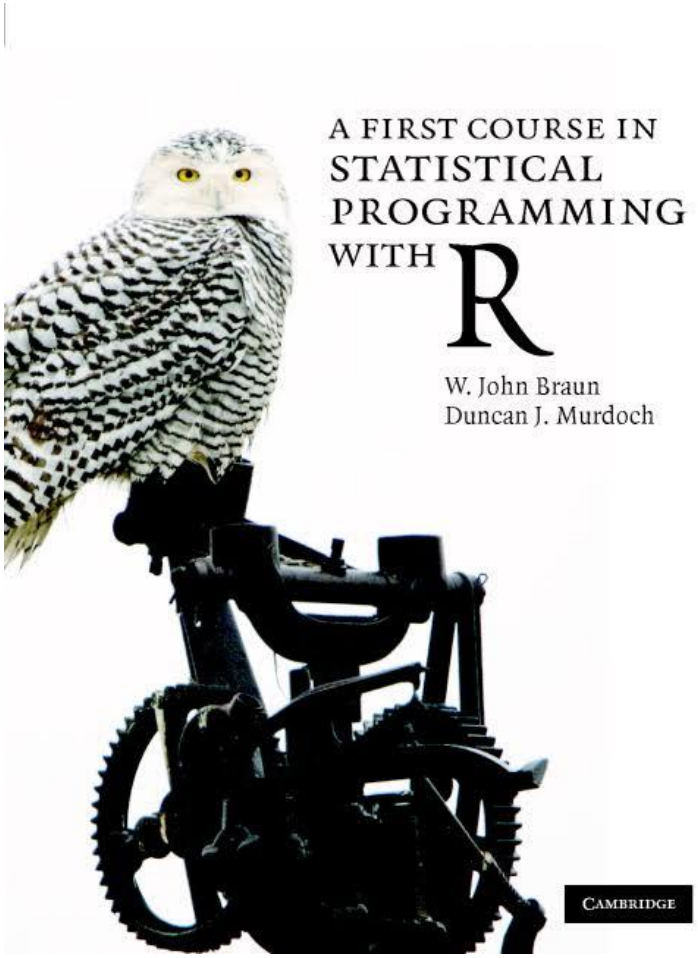
- Here, we have a function which will call itself. Something like recursive.factorial(x) will turn into x*recursive until x becomes equal to 0. When x becomes 0, we return 1 since the factorial of 0 is 1.
- This is the terminating condition and is very important. Without this the recursion will not end and continue indefinitely (in theory). Here are some sample function calls to our function.

```
> recursive.factorial(0)
[1] 1

> recursive.factorial(5)
[1] 120

> recursive.factorial(7)
[1] 5040
```

Useful recursive function

- The use of recursion, often, makes code shorter and looks clean. But it is sometimes hard to follow through the code logic. It might be hard to think of a problem in a recursive way. Recursive functions are also memory intensive, since it can result into a lot of nested function calls. This must be kept in mind when using it for solving big problems.

References


A FIRST COURSE IN
STATISTICAL
PROGRAMMING
WITH R
W. John Braun
Duncan J. Murdoch
CAMBRIDGE


THANK
YOU